IMM 380
April 1970

Courant Institute of
Mathematical Sciences

Individual and Multi-Processing Performance Characteristics of Programs on Large Parallel Computers

E. Draughon, J. Schwartz, and A. Stein

Prepared under Grant No. NSF-GJ-76
with the National Science Foundation

New York University

IMM 380
April 1970

New York University

Courant Institute of Mathematical Sciences

INDIVIDUAL AND MULTI-PROCESSING PERFORMANCE CHARACTERISTICS OF PROGRAMS ON LARGE PARALLEL COMPUTERS

E. Draughon, J. Schwartz, and A. Stein

Results obtained at the Courant Institute of Mathematical Sciences, New York University, with the National Science Foundation, Grant No. NSF-GJ-76.

Table of Contents

I. Introduction

As previously reported,[1,2] we have at the Courant Institute a Parallel Processor Simulator of an "Athene Class" computer which is able to simulate up to sixty CDC 6600 Central Processing Units each operating independently and all of which share a common memory; in addition, several instructions are provided which allow for intercommunication between the Central Processor Units. A Private Memory Feature which allows each Processor its own unique storage is also available (see the report [2] and particularly Appendix III of that report for detailed information on the Simulator).

Associated with the simulator we have a Fortran-like compiler with a number of Parallel Processing verbs and features (see [2] Section 5, and Appendix I, for details of the compiler) and an Operating system (described in [2], Section 6).

A preliminary report of our experience using the simulator was given in [2]; herein we will give a detailed account of our experiments with a variety of programs run under varied conditions and under the Operating system (often limited by the 6600 memory size).

We present graphs and tables of running characteristics, efficiency of utilization of processors and throughput as well as Operating System response time and similar information.

---

[1] J. Schwartz, "Large Parallel Computers," Journal of ACM, January, 1966.

[2] E. Draughon, et al., Programming Considerations for Parallel Computers, Courant Institute Report IMM 362, November, 1967.

In addition we summarize good and bad programming techniques; important features in Parallel Processing higher-level languages; system problems that turned up and our proposed solutions of them. We also compare our compiler language with the Tranquil language for the ILLIAC IV, and discuss the characteristics that make programs easily parallelizable.

II. The Simulated Operating System

The new features of our simulated operating system are reviewed first, as well as some deficiencies of the present system which would require correction in a production design. The timing measurements which were made are next described; then, by way of making the user's view of a parallel operating system more vivid, a number of common user errors are listed.

A. System features

The system is a multi-programming system providing priority, roll-in, roll-out, etc., as described in our earlier paper. We note the following key points.

1. The assignment of CPU's to jobs and the acceptance of them by jobs has not changed:[3] processors search sequentially through a priority list of numbered jobs; each job number on this list defines a slot on the job list which is examined using the NEWVAL instruction; if a negative value is returned, the processor returns to its search of the priority list; otherwise it takes a task from the task list and exits to that job.

2. Requests for a (virtual) number of CPU's or, equivalently, notification to the system by a job that a number of tasks are ready for parallel execution are still handled as previously (by placing the task address on the task list and incrementing the job list with a NEWVAL). However, in addition to the former

[3] Cf. E. Draughon, et al., Ibid., pp. 16-26.

request for an explicit number of (virtual) CPU's we now allow a request for a minimum number of CPU's. In case there are more idle CPU's in system status at the time of request than the number requested, the minimum number is used; if not, the number actually requested is used.

3. Roll-in and roll-out are simulated simply by replacing the job number of a rolled-out job with the number of the rolled-in job on the priority list.

Note that in our tests, the system had one job at each control point at "dead start" time. When each job finished, one of the remaining 9 jobs was "rolled in" to replace it. After the ninth job, any other job that had finished was "rolled in" again until each job had run several times.

4. It soon became necessary to add a job time limit feature, since some of the jobs never terminated.

5. No job purge was provided originally; but in order to remove all CPU's from jobs one of whose CPU's had made an error, and in order to reset pointers and clear lists for rerun of those jobs terminating normally, a purge was added to the system. A call to this facility (CALL TELOS) was also added for final termination, so that unanswered requests would be eliminated before the job was unloaded.

6. Some of the jobs turned out to be unable to accept new processors after a certain amount of time had elapsed. Since the user cannot know the actual number of CPU's he will get (as opposed to the number he requests), it became necessary to provide a function to remove or withdraw requested tasks.

This function was called WASHOUT. Since the task list is referenced in parallel and without any lockouts or waiting, the CPU which executes the WASHOUT must use NEWVAL and remove the tasks from the task list and job list one by one; a somewhat costly process.[4]

7. At "dead start" all CPU's except the one given to each job now go to a special wait loop where they divide themselves evenly among the number of jobs (or control points) in the machine. If no requests have been received after a short time, they enter the normal assignment loop and search the priority list.

This feature became necessary since in our original scheme the highest priority job tended to get all unassigned CPU's and the progress to completion of all other jobs suffered accordingly. Moreover, the situation never rectified itself: when the job with all the CPU's finished, the next job to be rolled in usually got them if they did not go into idle status.

---

[4] Most of this could be avoided by a total job WASHOUT indicator, thereby allowing an efficient retrieval of all the CPU'S but one for other jobs; but this technique would require that the indicator be referenced every time a request or retrieval is made by every processor.

B. Features desirable but not provided

1. No true core roll in - roll out has been implemented thus far.

2. No exchange jump is used in the system now. It appears that if the number of CPU's in the machine is less than the number of control points, a different kind of operating system which treats the CPU's collectively switching them between jobs (as in the CDC Chippewa system) might be preferable. At any rate, the present scheme can sometimes result in each job's either getting only one CPU (slowing job completion) or in certain jobs waiting for other jobs to finish (again slowing completion).

3. Sophisticated priority scheduling has not been used thus far: priorities are kept constant and no re-sorting of the priority list is done.

4. No I/O is simulated at present. A parallel I/O scheme has been coded and tested separately but has not been incorporated into the system as yet.

5. Memory protect is very difficult to implement as things now stand.

The central problem here is the need of a memory base address register in the simulator with resulting handling of requests in a location fixed relative to the base address.

Experience has indicated further features which parallel computer hardware and the operating system should have.

6. A job should normally not be rolled in unless there is more than one CPU available to service it. On termination of a prior job, it is likely that a number of CPU's will become available. Then, when a new job has been rolled in and given a processor, additional processor requests can be answered. It might also be possible to assign CPU's immediately at job roll in, using a "control card" type of request. The reason for this stricture on roll-in is that most (of our) programs had to withdraw their requests after a certain time. Thereafter, however, several CPU's would become available, would find no requests to answer, and would retire to idle status. The result was that several jobs were being executed with only one CPU while many CPU's were idle.

7. Job completion time will clearly be a function of the number of CPU's a job is assigned. This means that a meaningful user time limit feature must be stated in terms of the number of CPU's at work on a job.[5]

[5] That is, it should be arranged that the operating system keep a running count of the total time of all the CPU's executing a given job and make its time limit decisions on this basis of total CPU execution time.

8. For the user's and the installation's protection, some control of the amount of idle time which a given user can accumulate is necessary. Programmer errors and sloppy coding can lead to the accumulation of enormous quantities of purely idle time.

Such a check was actually included as part of our simulator, but did not represent any specified hardware. If the total accumulated idle time for all jobs in the simulated system became greater than 50%, the entire run was terminated. This has been one of the most frequently used modes of termination.

In a real machine it is not clear how this check should be implemented, but its necessity is obvious.

9. As the system is now designed, it has turned out to be somewhat unresponsive to user requests, simply because all CPU's are almost always being used by jobs.

One solution of course is to allow a requesting CPU to "steal" other CPU's from other jobs via an exchange jump. But unless there is frequent and perhaps unanimous switching of CPU's from a given job, it seems very likely that the jobs would fail to execute properly if a processor were stolen from them, for: (1) Some jobs keep records of the number of CPU's which have been assigned to them and they commit errors if they do not have that many. (2) It is possible that some CPU's, other than the one stolen, are in an idle loop waiting to be released by the stolen CPU. (3) Processors may

go into an idle loop waiting for all their fellow processors to finish a piece of code (including the one stolen).

Thus uncoordinated interjob exchange of processors is not really feasible.

C. Timing measurements available in the simulated system

The following quantities are measured in the simulated operating system.

1. Completion time. This measures the period extending from the time the first CPU enters a job through the job purge at the end of execution. No I/O time is included.

2. Total processor-use accountable to each job. This includes the total number of cycles (instructions) executed by all CPU's which each job utilized. It is split into useful time (time in which the code actually applied to the solution of the given problem) and idle time (spent in wait loops). It is up to the problem programmer to tell the system into which of these categories the various sections of his program fall.

3. Task-request time. This measures the number of instructions executed in setting up new tasks or in other words in requesting more CPU's.

4. Response times to requests. This is the number of elapsed cycles between the time a request is placed on the task list and the time a CPU picks up the task from the list.

5. System time and idle time. System time is the total number of cycles spent by all processors in executing system code. The system idle time in this case is the total number of cycles of code executed in the system wait loop and thus gives an indication of how efficiently all the jobs put together were utilizing the machine. These quantities depend in turn on (1) the number and frequency of requests by simulated jobs, (2) job mix, (3) other circumstances (e.g. if a job has WASHOUT-ed some requests just before several CPU's become available, they may become idle).

6. The number of idle CPU's. Each time any CPU's become idle they count themselves and this number is saved to get a maximum, minimum, and average.

7. The number of unsatisfiable tasks. Whenever there are more requests than CPU's the number is saved; ultimately the maximum, minimum, and average of this quantity is obtained.

8. Percentages of time falling into various categories. The percentage of useful, idle, etc., time is measured periodically, and normalized to 100% both cumulatively and for a particular period.

9. Percentages of CPU's in jobs, in the system, or in system-idle status at a given moment. These percentages were initially measured periodically; eventually this measurement was discontinued since the CPU's were usually all busy in jobs,

and since the effect it was designed to measure seemed to be well covered by measures 8 and 10 .

10. Ratios of useful to idle, useful to system and useful to all other types (called "useless") were measured at the same periods at which measures 8 and 9 were obtained. These ratios are of course measures of overall system (and machine) efficiency.

To estimate the extent to which the various measurements discussed above are dependent on certain other parameters of the system and the machine, the following deliberate parameter variations were made:

(1) Number of CPU's. Machines with 5, 10, 15, 20 and 25 CPU's were simulated.

(2) Number of concurrent jobs. Four and five concurrent jobs were tried.

(3) Number of tasks requested by each job. An attempt was made to cause these numbers to vary, but the variations seemed to have little effect except in conditions of "underload" (too few requests), or "overload" (too large an accumulation of requests). In the normal middle ground this parameter seemed not to play much of a role.

D. Errors detected by the simulator

The simulator checked for errors, stop instructions, etc., and in addition caught errors in user requests to the system:

1. Too many: Total accumulated number of tasks was limited by the size of the circular task list to 1000_8.

2. Incorrect request: Requesting zero or fewer tasks was not permitted by the system; task addresses out of range were also caught by the simulator.

E. Common difficulties in using the system

To make the user view of our hypothetical system more vivid, the following problems encountered in use are cited.

1. Some programs broke down in various ways when some CPU's answered their requests much later than other CPU's. Other programs put such late-comers into idle loops. If such a limitation is "inherent" in a program, the program should check the time of arrival of all CPU's and return unused ones. The newly provided WASHOUT feature (see above) is intended as an aid in this connection.

2. Programs frequently put CPU's into idle loops for long periods rather than returning them.

3. Some jobs returned all CPU's while still having tasks outstanding on the request list. In one case, this was simply a failure to call "TELOS" and get the job purged at the end. In another case the job was not finished and the requests were meant to be answered. This is simply poor programming. Instead of being returned, the CPU's should be transferred to the start of unfinished tasks.

4. Sometimes mal-coordination puts all processors into idle loops leaving none left to wake them up. This error is more easily committed than one might expect.

5. Some of the programs used algorithms correct only for two or more CPU's. Attempting to execute them with only one processor assigned led to errors.

6. Failure to declare arrays "public" and underestimation of the number of private variables sometimes resulted in job abortion, since space for private variables is assigned as needed rather than in advance.

7. It often happens that a job commences when almost all of the processors are engaged, thereby forcing the new job to operate with one or only a few processors. The problems inherent here might be addressed by providing either priority retrieval, an ability for jobs to lockout retrieval attempts, a method for judging which processors would be most economical to retrieve, or an indication to a job when processors were retrieved from it by the system.

We have found that quite often a run-time decision had to be made as to how to apportion processors to jobs requesting them at the same time. We solved the problem by essentially using a first-come first-served method but clearly this is inadequate. What is needed is a decision routine either supplied by the user for each job or a generalized one which would take account of priority and expected gain per additional processing unit. The difficulty here is that the CPU's may spread themselves too thin over the jobs so that no job attains optimum efficiency. Appendix III below outlines an alternative system which might be better able to cope with these problems.

8. One may sometimes wish to use algorithms which are such as to require a foreknowledge of how many processors will be available during an actual run. It would be desirable if the operating system could predict the number of processors that will be available within a reasonable amount of time. Such a prediction should at least involve a count of all processors involved in WASHOUT and TELOS operations. (See discussion above). A more refined solution could be provided using a system option allowing an indication of imminent completion of jobs by processors executing them. In the current implementation we used the following poor expedient for this same purpose: a delay routine which allowed a measured delay of the processor executing it, thus giving jobs the ability to wait a reasonable time for the arrival of processors. All processors arriving later are then sent back to the operating system. On the other hand, if jobs were only rolled in when there were enough CP's available to execute them optimally, throughput might suffer considerably. See also our subsequent discussion of an alternative operating system in Appendix III.

9. Many routines periodically send processors back to the operating system and request other processors at a later time; this makes necessary a method for retrieval of any extensive private storage (i.e. storage uniquely reserved for each processor) existing at the time of processor exit. Since processors returned by the system are selected at random this means that the unique number (badge number) of the processor

to be used should be set on job entrance. It follows that revisions are necessary in the presently proposed Read Badge Number instruction (see [2], p. 7). In general, situations in which data in private storage must be saved for later use imply restrictions on the number of processors that can usefully be accommodated in late execution phases of a program; processor calls must reflect this. I.e., the number of parallel parts a program opens to begin with may in some cases set an upper (and perhaps a lower) limit on the number of parallel parts later sections may have. In an operating system like the present one, this again implies run-time adjustment of algorithms.

III. Comments on Parallel Programming Techniques

The experience of our group in programming our repertoire of routines and in programming the simulated operating system has made us aware of a number of desirable techniques and of some particularly bad ones. We give some interesting and reasonably general examples.

A. Programming pitfalls

1. Computations that can be combined with a slight amount of effort should not be separated. This mistake characterized our least successful "parallelization", the complex eigenvalue computation; the programmer consistently coded the parallel sub-tasks completely independently of each other in the sense that only the smallest DO-loops were individually parallelized (involving substantial parallel overhead). Still worse, this meant that groups of processors continually had to wait at the end of a parallel segment until the last member of the group completed a loop. Note the following example taken from the code (actually somewhat modified; the original was even worse).

```
      DOP (60) 40  K = 1,N [6]
      A(K,J) = DIV * A(K,J)
      IF (K.GE.I) A(J,K) = A(J,K)/DIV
40    CONTINUE
      SUM = 0.0
      IHOLD = 0
      GO TO 70
60    IF(IHOLD.NE.0) GO TO 60
70    DOP(6) 75  K = 1,N
75    CALL FRAD(SUM, A(K,J) * CONJG(A(K,J)))
      [etc.]
```

which is equivalent to the much simpler and more efficient

```
      SUM = 0.0
      DOP (60) 40  K = 1,N
      IF (K.EQ.J.AND.J.GE.I) GO TO 40
      A(K,J) = DIV * A(K,J)
      IF (K.GE.I) A(J,K) = A(J,K)/DIV
40    CALL FRAD(SUM, A(J,K) * CONJG(A(K,J)))
      [etc.]
```

---

[6] DOP(S_1)S_2 I = IN,FIN,ST is our form of the parallel DO-loop instruction (see [2], p. 13), with the meaning: Assign in parallel a different value of I for each executing processor ranging from IN through FIN by ST; send all processors whose value of I would exceed FIN to statement S_1 and send the last processor to complete the loop to the statement following S_2.

2. Almost parallel matrix manipulations should not be serialized. In the following example the switching of two rows and columns was done serially whereas the intersection clearly should have been saved and restored at the end of the loop.

```
      DOP (12) 11  KK = 1,N
      CC = A(J,KK)
      A(J,KK) = A(NSUB,KK)
11    A(NSUB,KK) = CC
      IHOLD = 0
      GO TO 13
12    IF (IHOLD.NE.0) GO TO 12
13    DOP (15) 14  KK = 1,N
      CC = A(KK,J)
      A(KK,J) = A(KK,NSUB)
14    A(KK,NSUB) = CC
      [etc.]
```

3. The execution of a series of tasks should not be delayed until a previous series of tasks is completed when some of the following tasks can be executed without delay. This error occurred, e.g., in a sort routine, in which all the substrings were sorted before the following merges were started, resulting in considerable inefficiency.

4. Errors of the preceding type have been found not only to reduce computer utilization efficiency but to cause numerous debugging difficulties (since the interdependence between CP's is increased) and should be avoided whenever possible.

5. Portions of parallelizable code which contribute little to the overall job throughput time should be left serial.

6. The general comment can be made that algorithms that exhibit little parallel structure should not be parallelized inasmuch as the small gains in single-job completion time will be more than offset by the large losses in overall system throughput.[7]

B. Desirable techniques

1. Our best results were obtained with programs of the following kind:

(a) Programs containing parallel DO-loops, i.e., programs in which all processors execute almost the same instructions, much as they do in the Illiac IV.

(b) Programs containing parallel semi-independent sections, wherein the processors execute similar algorithms but where variations in the data could cause quite different local orders of execution (as in sorting, for example).

(c) Programs decomposable into almost independent tasks, wherein the processor execute completely different sub-tasks whose results are then consolidated at some later point.

It is clearly desirable to force the time-consuming sections of algorithms to be parallelized in some such optimal fashion; and this can often be done without great difficulty.

---

[7] Again, this clearly applies only to Athene systems.

2. The portions of routines that are to be parallelized should be examined with regard to the type of data that must be operated on. For example, in the parallel shooting algorithm for the solution of two-point boundary value problems (see below for additional details), if the dimension of the set of linear equations in part two (see p. 22) of the algorithm is large that part of the program should be parallelized, otherwise not. In other examples (e.g. matrix multiply), different configurations of data, as e.g. a 2 × 100 matrix as against a 100 × 2 matrix, may require different parallelizations.

3. The number of processors expected to execute an algorithm will often determine the portions of the program which are worth making parallel (again note the parallel shooting case). E.g., if there are many CPU's, the consolidation of final results bulks large and should be parallelized, otherwise it need not be.

The actual number of processors which will become available will not be known before run time unless the control system is such as to guarantee the delivery of the requested number of CP's. It may therefore even occasionally be advisable to change algorithms for different numbers of processors (compare our parallel search which has quite different algorithms for odd and even numbers of processors).[8]

4. Queuing. Our experience has shown that each queuing point in a program should be examine with a view to ameliorating queuing delays, always keeping in mind both program flow and the number of processors that may be available for the program's

[8] E. Draughon, et al., p. 29.

execution. (In programs with limited queuing the effect of queue delays on completion time and execution efficiency only becomes apparent if a large number of processors are used).

5. Random numbers. A possible difficulty in random number generation for a Monte-Carlo program was eliminated by providing each processor with its own starting point in the random series produced by the generator; these starting points were separated by large constants to avoid any side effects that might be caused by duplication of the random numbers in independent processors.

6. Although a general scheme allowing for semi-private storage through use of a parallel compiler is desirable (see our comments on the Tranquil language in Section IX), no such facility is available in our simulated system. In several cases, we mimicked the effect of such storage. The following are typical of the techniques used.

a. Portions of a public array were temporarily allotted to individual processors for storage of semi-final results which were subsequently combined into final results in a "public" manner.

b. Private subparts of public words were occasionally assigned to different processors and later processed as public variables. This requires use of the replace add instruction in conjunction with shifts. Subfield operations cannot be done in parallel on the bits of a public word except by locking out the public word in question.

c. Private information on public arrays was sometimes kept and the arrays accessed via private pointers generated from a public variable using the replace-add instruction. This technique allows new processors to continue a job at points where previous processors were discontinued. For example a pseudo-badge number may be generated for each CPU entering a job; this badge number may then be saved when a CPU exits so that any other CPU entering can continue the role of the CPU which exited.

IV. Summary of Programs Simulated in Our Experiments, and Associated Experience

We will describe the more interesting programs we have programmed and run, mentioning some of their characteristics; for a discussion of others, see [2], pp. 28-30.

A. Parallel shooting

This is a method for solving two point boundary value problems for ordinary differential equations;[9] it involves two steps:

1. A parallel integration where the interval of integration is divided into N equal parts, N being the number of processors.

2. Solution of a set of linear equations in which the results of the above integration are used; these equations define increments to a starting value vector, allowing iteration.

We programmed this algorithm so that only step 1 above was done in parallel, a procedure found to be reasonable if there are not too many processors. However, our results show that the computation time required for the second part of the algorithm rises quite rapidly with increase in the number of processing units, so that with many processors it would also be desirable to use a parallel algorithm for the solution of the linear equations. Our data on this program show the result of running the program with varying termination criteria, resulting in varying numbers of iterations performed

[9] Due to H. B. Keller.

in finding the solution. This is a program in which the efficiency of processor utilization depends strongly on the input data.

5. Computation of total potential energy of a system of N particles interacting by pair-wise Lennart-Jones forces

The program consists of a series of nested loops; the uppermost three were in effect combined into one large parallel do-loop using the DOP instruction of our parallel compiler PFORTRAN[10] (see [2], pp. 12-16 for a description of the PFORTRAN parallel verbs) and by assigning the actual subscripts using an algorithm. Inner iterations were thus shared with complete independence between processors; the final contribution of each computation to total energy then being accumulated sequentially using the PFORTRAN LOCK and UNLOCK instructions (Cf. [2], pp. 14-15). (Since this is only a miniscule portion of the computation, the procedure used does not affect the efficiency of the program). In the simulated operating system environment this program had the property of eventually absorbing all the processors it could handle, as this total was initially requested, and as processors were released by other jobs they were assigned this job. The behavior of completion time for this program, as indicated in the tabular summary presented below,[11] is worth noting.

---

[10] This is equivalent to the substitution of a single cartesian product incorporating the otherwise separate indices of the multiple do-loops; a procedure that is more elegantly accomplished in the Tranquil language by use of a cross-product index set (see Appendix II for a critical discussion of the Tranquil language).

[11] Cf. Chart VIII below.

C. Monte Carlo computation of atomic energy levels ([2], p. 30).

This program executes a large number of random subprogram paths which result initially in the storage of preliminary results in private arrays and finally in the updating of public arrays from which final results are computed. This program is interesting in that it represents an example of a program midway in characteristics between these exhibiting DO-loop type and a FORK type of parallelism (Cf. the remarks of Dr. B. Lambson in Section G/2 of the 1969 Spring Joint Computer Conference). In parallel programming of the former type, all the processors perform almost the same operations (a situation eminently adapted to the Illiac type of parallel processor).[12] In parallel programs of the latter type the processors perform essentially unrelated procedures. In our situation the processors execute similar but quite different program segments. (See below for a summary of the various types of parallelism exhibited in our repertoire of programs).

Several points may be noted in connection with this program.

i. A previous version of this program, reported in ([2], pp. 36,38-9), exhibited queuing delays when a large number of processors were used. These delays occurred in the generation of random numbers. This problem was rectified, with a noticeable improvement in performance, by arranging the independent computation of series of random numbers for each of the executing computers.

[12] Athene type processors can ignore the difference between these two types of parallelism as far as the hardware is concerned, as they are capable of performing completely independent processes simultaneously.

ii. As indicated, the program consists of a parallel series of random computations followed by a calculation of final results. This final step is an essentially serial and moderately lengthy procedure. It is therefore reasonable in the final part of this program for one CPU to interrupt the operations of the other processors and return them to the system. As an interrupt was not specified in our simulated hardware, we used instead a public communication variable to be tested periodically but in a manner which would not significantly increase the running time. Our experience with this technique has shown that in cases resembling the current case it is quite as effective as an interrupt would be, i.e. not notably more costly in execution time.

iii. In order to be able to continue the processing of private tasks past a point of suspension and system return, each entering processor was caused to generate its own pseudo-badge number and to use it for reference of public arrays. When a processor re-entered, it simply re-generated a pseudo-badge number (perhaps a different one) and resumed processing. Thus, any n processors could be returned in any order to the system; it was unnecessary to recover the same CPU's that had been given up. This type of badge number assignment is superior to the absolute badge number type, and suggests a modification of the Read Badge Number instruction presently specified for our simulated hardware.

D. Computation of shockwave characteristics

This routine was adapted from a program of Dr. A. Rotenberg of the Courant Institute. The algorithm involved a large number of nested DO-loops independent at the highest level, allowing trivial parallelization; this "parallel" portion of the program was followed by a largely linear treatment of special cases. When run by itself using large numbers of processors, the program tended to run rather inefficiently since the handling of special cases consumed a considerable portion of the running time. However, in the operating system environment the program ran well on our simulated Athene type computers as is demonstrated by a relative efficiency graph, shown below.

E. Calculation of eigenvalues of a complex matrix.

This program used an algorithm based on the QR method[13] and was adapted from the library QREIGEN subroutine[14] of the Courant Institute Computing Center.[15] It was parallelized by assigning matrix row and column operations to separate processors. Unfortunately, the programmer controlled individual processors more tightly than necessary, with the result that speed increased only poorly on the addition of more processing units. A 10% increase in efficiency was then gained by the simple expedient of treating a particularly outrageous portion of the program linearly.

---

[13] Frances, J. G., 1961, 1962, "The QR Transformation," Parts I & II, Computer Journal 4, 265-271, 332-345.

[14] Murray, Richard. NYU Master's Thesis, June 1967.

[15] New York University, AEC Computing Center Handbook, Sect. 8.

V. Quantities Computed for Programs Run Outside the Simulated Operating System

Our program measurements are reported in terms of a number of quantities derived from simulations.

A. Normal overhead

This is taken to be

$$O = \frac{\Sigma\ (T_{com} + T_s)}{N}$$

where

T_{com} is the time required for parallel communication;

T_s is the set-up time for parallelization;

N is the number of processors;

the sum above extends over all processors.

It is clear from the definition that we have a lower bound on overhead, inasmuch as we here assume zero time for return of the idle processors to the system and their retrieval for any necessary later use; however, our experiments have shown that even for our simple system the difference between our O and other related measures of overhead is not large.

B. Efficiency

This is taken (as in [2], Section 10) to be

$$E_N = \frac{T_1}{N_A * T_C} = \frac{T_1}{\Sigma\ T}$$

where

T_1 is the time required to run the program on a serial machine;

T_C is the program completion time when run on our parallel machine; and where

$N_A = \Sigma T / T_C$.

the sum accumulates the total time spent by processors in the execution of the job.

C. Relative efficiency

This quantity gives an upper limit for the efficiency of parallel processing of single jobs. It is defined as

$$E_R = \frac{T_1}{N * T_C - \sum T_R}$$

where

$\sum T_R$ is the sum extended over the total recoverable[16] idle time;

N is the number of processors.

---

[16] It should be noted that the distinction we make between recoverable idle time and irrecoverable idle time is applicable only to Athene type computers with multi-processing operating systems as distinct from machines of Illiac IV or CDC Star types. For these latter machines the results of single-job simulations will not differ from run measurements made within an operating system, since there is no recoverable idle time (in this sense). Of course, if an Athene user refuses to give up his CPU's to the system, the distinction also vanishes.

D. Cost effectiveness

To get an idea of this we have used the somewhat arbitrary definition of M. Lehman[17] normalized to 1 in the case in which only one CPU is in operation. Our formula is

$$Q = \frac{T_1^2}{T_C \times \sum T}$$

where T_1, T_C and $\sum T$ have the significance already explained.

[17] Lehman, M. "Proceedings of IEEE," Vol. 54, p. 1899.

VI. Measured results for Programs Run Singly and Under the Operating System.

We broke our study of parallel program characteristics into two phases: measurement of parallel programs run individually using varying numbers of processors and sets of data, and measurements referring to simultaneous execution of a number of jobs run under the control of our operating system. In general we have tried to distinguish useful time[18] from parallel set-up time. Where appropriate we have even, when jobs were run singly, attempted to distinguish recoverable idle time (i.e. idle time in blocks of length sufficient to make it economical to send the idle processing units back to an operating system) from irrecoverable idle time. In simulated runs under the operating system (reported in a subsequent section) we simulated various mixes of jobs and included both a kind of priority system and a first-come first-served system in our experiments.

The following terms are used in the charts that follow:

Completion time. Time for completion of job measured from initial entry into parallel mode until the final exit.

Efficiency. Ratio of completion time for one CPU to the product of the number of CPU's and the completion time.

Total cycles. Total CPU time assuming no CPU retrieval (multiprogramming) even when possible.

---

[18] This may not be actually useful in that the CPU's need not be executing code which contributes in any higher sense to the solution of the problem; we only detect that they are not in idle or system status.

Relative efficiency. Ratio of completion time for one CPU to the total CPU time assuming CPU retrieval where possible.

Total irretrievable cycles. Total CPU time assuming retrieval where possible.

Q-cost effectiveness (relative). See text; assuming CPU retrieval is allowed.

Q-cost effectiveness (total). Assuming CPU retrieval is not allowed.

N-average. The "average" number of CPU's performing useful work during the run; see text.

O-normal overhead. Time necessary for parallel setup and communication.

SHOCKWAVE COMPUTATION

(Individual Runs with Two Input Data Configurations,

Measurements in Simulated Cycles)

	No. of CPU's	Completion Time	Total Cycles	Eff.	Total Irretrievable Time	Rel. Eff.	Normal Overhead	Q Relative	Q Total	N Average
Run 1	1	4.01×10^5	4.01×10^5	1.00	4.01×10^5	1.00	0.0×10^2	1.0	1.0	1.0
	2	2.26×10^5	4.52×10^5	.888	4.01×10^5	1.00	0.0×10^2	1.8	1.6	1.8
	10	8.57×10^4	8.57×10^5	.468	4.02×10^5	.998	1.0×10^2	4.6	2.2	4.7
	20	6.77×10^4	1.35×10^6	.296	4.12×10^5	.975	5.5×10^2	5.8	1.8	6.1
	30	6.19×10^4	1.86×10^6	.216	4.14×10^5	.968	4.3×10^2	6.3	1.4	6.7
	40	5.91×10^4	2.36×10^6	.170	4.18×10^5	.960	4.2×10^2	6.5	1.2	7.1
	50	5.65×10^4	2.82×10^6	.142	4.31×10^5	.930	6.0×10^2	6.6	1.0	7.6
	60	5.39×10^4	3.23×10^6	.124	4.17×10^5	.963	2.6×10^2	7.1	.9	7.7
Run 2	1	2.67×10^5	2.67×10^5	1.00	2.67×10^5	1.00	0.0×10^2	1.0	1.0	1.0
	2	1.51×10^5	3.01×10^5	.887	2.67×10^5	1.00	1.0×10^2	1.8	1.6	1.8
	4	9.22×10^4	3.69×10^5	.721	2.67×10^5	1.00	1.0×10^2	2.9	2.1	2.9
	8	6.30×10^4	5.04×10^5	.529	2.67×10^5	1.00	1.0×10^2	4.2	2.2	4.2
	16	4.84×10^4	7.75×10^5	.345	2.68×10^5	.996	1.1×10^2	5.5	1.9	5.5

CHART I

The matrices used on the set of runs labelled Run 2 were half the size of those in Run 1. The decrease in completion time with increaess in number of CPUs (first and second columns) is evident. The remaining columns show the measures explained on the preceding page.

Additional Experiments with the

SHOCKWAVE COMPUTATION

Showing Runs Under the Operating System

No. of CPU's	Completion Time	Total No. of Cycles	No. of Control Points	No. of CPU's in Machine
7	161,125	359,306	4	30
7	203,922	365,979	4	25
12	147,791	365,789	4	20
12	145,980	359,365	4	25
16	147,035	364,169	4	25
16	148,021	364,501	4	20
18	143,843	365,709	4	30
21	138,982	352,942	4	30

CHART II

The runs in this example were made under the operating system. Since all the CPU's did not arrive at the job at the same time, the completion time varies even when the number of CPU's (column 1) is the same. This arrival time was of course affected by the number of CPU's in the machine; in general the fewer CPU's in the machine the longer the arrival time.

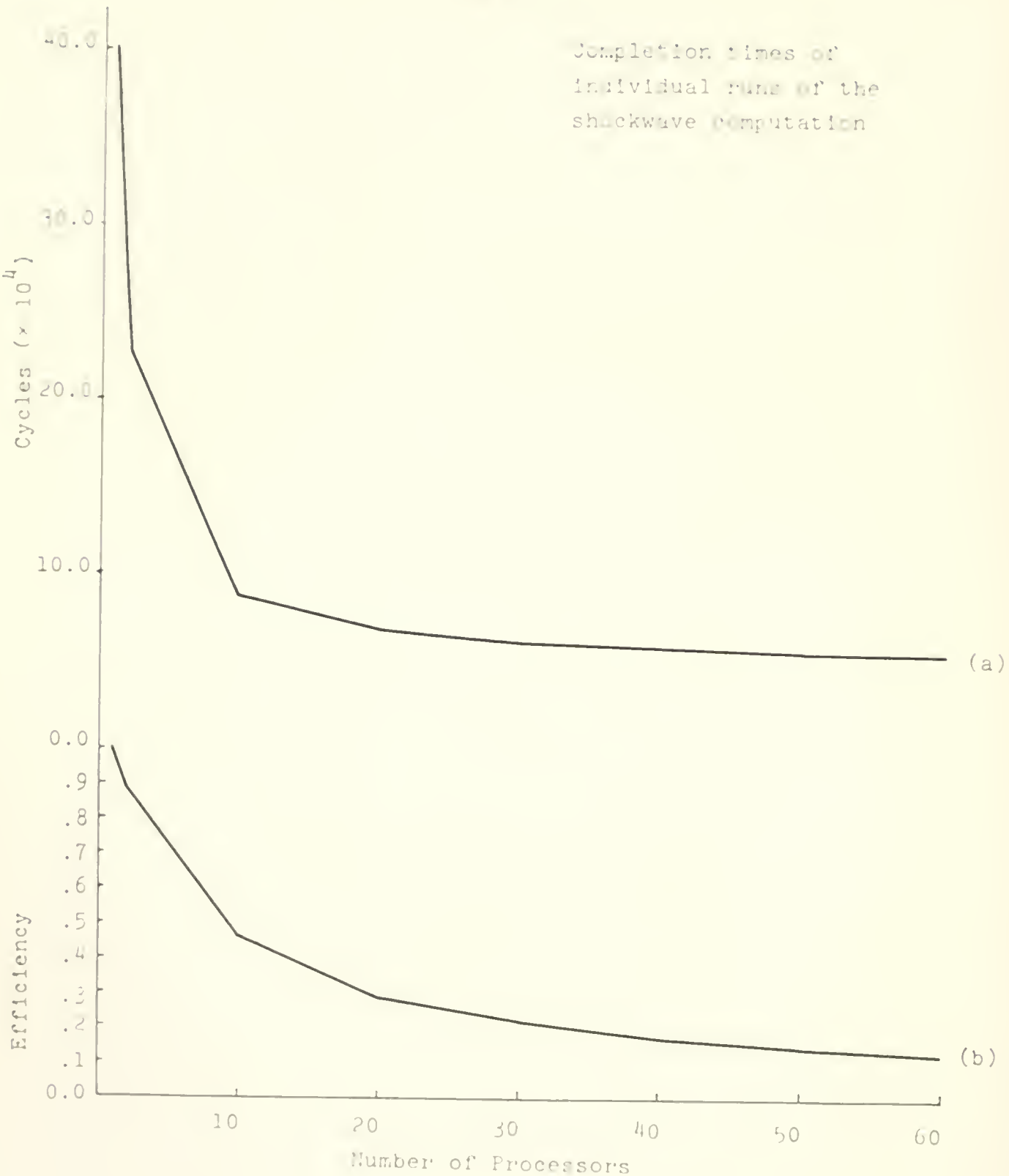
The data used in these runs was the same as that reported for Run 2 in the previous chart on the shockwave program. Yet execution took more than twice as long! This is presumably due to the fact that CPU's could arrive at any time after the start of the program.

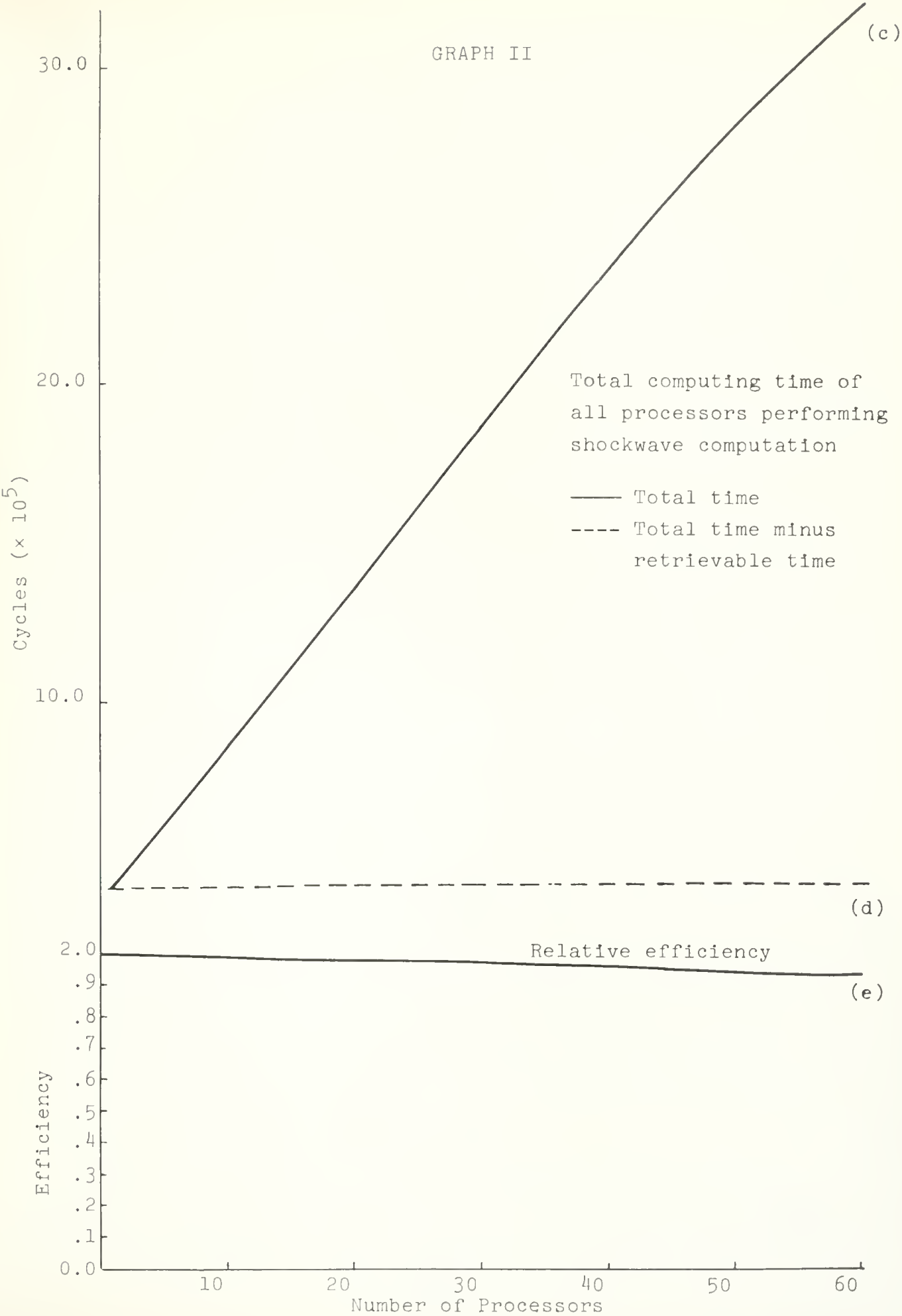
The data for the set of runs labelled Run 1 on the previous chart is represented in graphs I and II.

In graph I, curve (a) is the completion time, curre (b) is the efficiency (column 4 of the chart).

In graph II, curve (c) represents the total cycles (column 3 of the chart), curve (d) is total irretrievable time (column 5), curve (e) is relative efficiency (column 6).

GRAPH I


Completion times of individual runs of the shockwave computation
40.0
30.0
20.0
10.0
Cycles ($\times 10^4$)
(a)
0.0
.9
.8
.7
.6
.5
.4
.3
.2
.1
0.0
Efficiency
(b)
10
20
30
40
50
60
Number of Processors


GRAPH II
(c)
30.0
20.0
10.0
Cycles (× 10^5)
Total computing time of all processors performing shockwave computation
—— Total time
---- Total time minus retrievable time
(d)
2.0
.9
.8
.7
.6
.5
.4
.3
.2
.1
0.0
Efficiency
Relative efficiency
(e)
10
20
30
40
50
60
Number of Processors

COMPLEX EIGENVALUE COMPUTATION

8 × 8

With Two Program Configurations

	No. of CPU's	Completion Time	Total Cycles	Eff.	Total Irretrievable Time	Rel Eff	Normal Overhead	Q Total	Q Relative
Run 1	1	68,018	68,018	1.00	68,018	1.00	16×10^2	1.0	1.0
	2	44,290	88,580	.77	73,243	.93	6.7×10^2	1.2	1.4
	3	36,953	110,859	.61	79,386	.86	4.5×10^2	1.1	1.7
	4	32,401	129,604	.52	86,171	.79	3.4×10^2	1.1	1.6
	5	31,406	157,030	.43	93,398	.73	2.7×10^2	.9	1.7
	6	30,683	184,098	.37	99,752	.68	2.2×10^2	.8	1.5
	7	28,801	201,607	.34	107,448	.64	1.9×10^2	.8	1.4
	8	27,862	221.896	.31	108,318	.63	1.6×10^2	.7	1.5
Run 2	1	66,453	66,453	1.00	66,453	1.00	16×10^2	1.0	1.0
	2	43,957	87,914	.76	73,457	.90	6.9×10^2	1.1	1.4
	3	36,458	109,374	.61	79,706	.83	4.6×10^2	1.1	1.5
	4	31,826	124,304	.54	83,534	.80	3.5×10^2	1.1	1.7
	5	30,845	154,225	.43	93,987	.71	2.8×10^2	.9	1.5
	6	29,992	179,952	.37	98,977	.67	2.3×10^2	.8	1.5
	7	28,179	197,253	.34	108,297	.62	2.0×10^2	.8	1.4

CHART III

The second set of figures (Run 2) represents the results obtained when some of the more gross programming defects present in the first version of the program (Run 1) were removed; see text.

EIGENVALUE CALCULATION FOR COMPLEX MATRIX

8 × 8

Runs Under the Operating System

No. of CPU's	Completion Time*	Total Useful Cycles*	Total Idle Cycles*	No. of Control Points	No. of CPU's in Machine
1	76	73	0	4	20
2					
3	43	85	39	5	20
4					
5	38	103	79	4	20
6					
7	37	118	126	4	20
8	35	124	142	4	20

*In thousands.

CHART IV

Comparing these completion times under the operating system with those for individual runs on the previous page, one notes (1) that for 3 CPU's the execution time under the system is 11% longer, and that (2) this difference in execution time gradually increases to 25% for 8 CPU's. So, the cost of the operating system to the individual program increases rather dramatically with increases in the number of CPU's.

These results were obtained using the improved version of the eigenvalue program.

Further Examples of the

COMPLEX EIGENVALUE CALCULATION

	No. of CPU's	Completion Time	Total Cycles	Eff.	Total Irretrievable Time	Rel. Eff.	Normal Overhead	Q Total	Q Relative
Run 1 (6 × 6)	1	132,037	132,037	1.00	132,037	1.00	12×10^2	1.0	1.0
	2	116,537	233,074	.57	142,667	.92	5.4×10^2	.6	1.0
	3	112,500	337,500	.39	154,308	.86	3.7×10^2	.4	.9
	5	109,120	545,600	.24	177,216	.75	2.1×10^2	.3	.9
	6	109,487	656,922	.20	184,432	.70	1.8×10^2	.2	.8
Run 2 (4 × 4)	1	28,241	28,241	1.00	28,241	1.00	6.9×10^2	1.0	1.0
	2	23,186	46,372	.61	32,973	.82	2.9×10^2	.7	1.0
	3	21,144	63,432	.44	38,141	.74	2.0×10^2	.6	1.0
	4	20,637	82,548	.34	43,140	.65	1.4×10^2	.5	.9

CHART V

In these two sets of runs, the improved version of the program was run on different data: the resulting decreases in completion time with increases in number of CPU's is apparent in each case.

N-PARTICLE POTENTIAL ENERGY COMPUTATION

No. of CPU's	Comple-tion Time	Total Cycles	Eff.	Total Irretriev-able Time	Rel. Eff.	Q Total	Q Rela-tive	N Aver-age
1	562,963	562,963	1.00	562,963	1.00	1.0	1.0	1.0
2	284,065	568,130	.99	563,055	1.00	1.9	2.0	2.0
3	213,268	639,804	.88	563,147	1.00	2.4	2.7	2.6
4	144,646	578,584	.97	563,239	1.00	3.8	4.0	3.9
5	143,035	715,175	.79	563,331	1.00	3.1	4.0	3.9
6	142,808	856,848	.66	563,423	1.00	2.6	4.0	3.9
7	142,583	998,081	.56	563,515	1.00	2.2	4.0	3.9
8	73,612	588,896	.96	563,607	1.00	7.0	7.7	7.7

CHART VI

Here is an example of a program having characteristics ideal for parallelization (note the efficiencies for 2, 4, 8 CPU's) where the actual utilization of processors could fall almost 50% for a parallel processor of the Illiac IV type with a number of arithmetic units mismatched to the data.

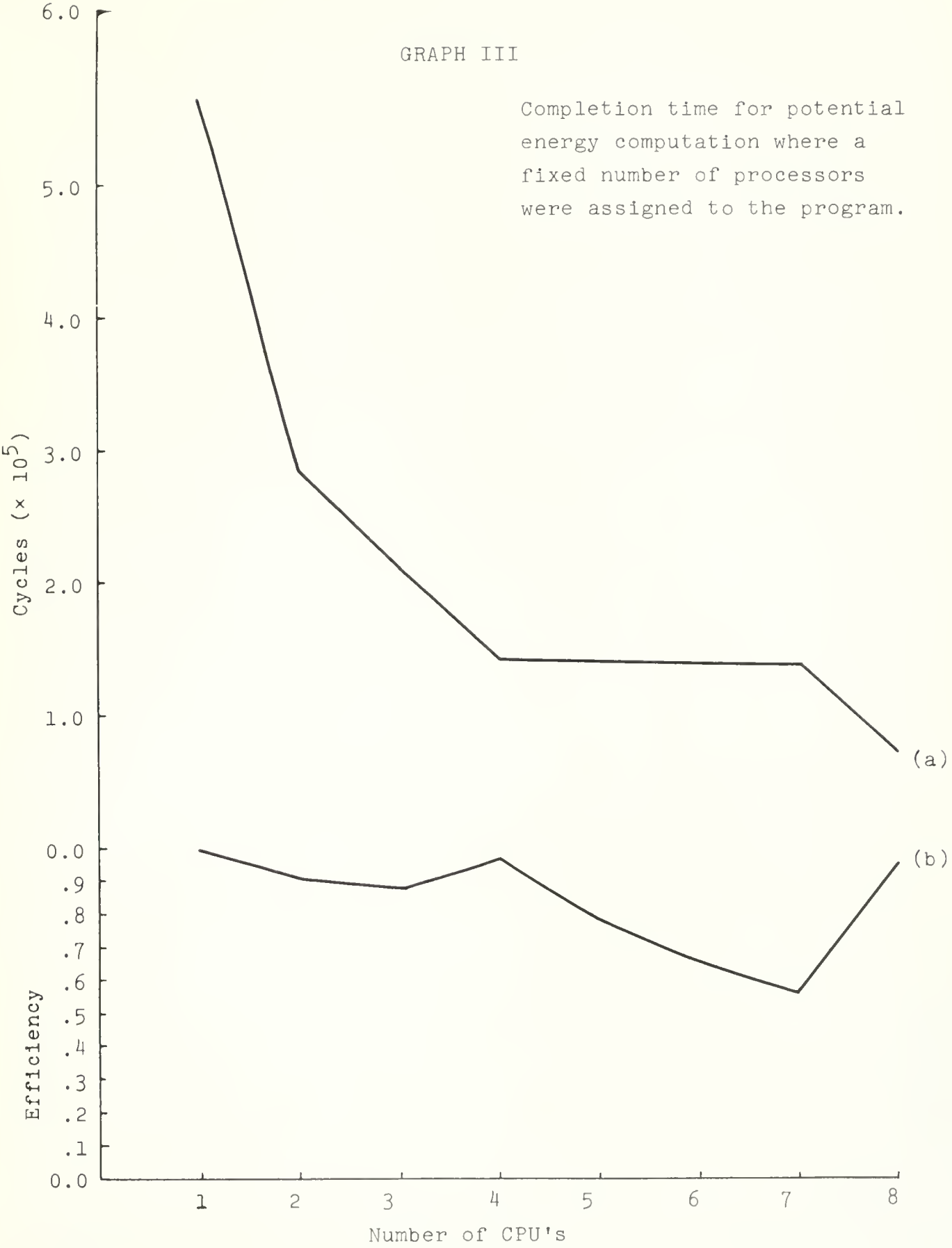
The graphs on the following two pages show the data from the preceding chart.

Curve (a) is completion time (column 2 of the preceding chart); curve (b) is efficiency (column 4).

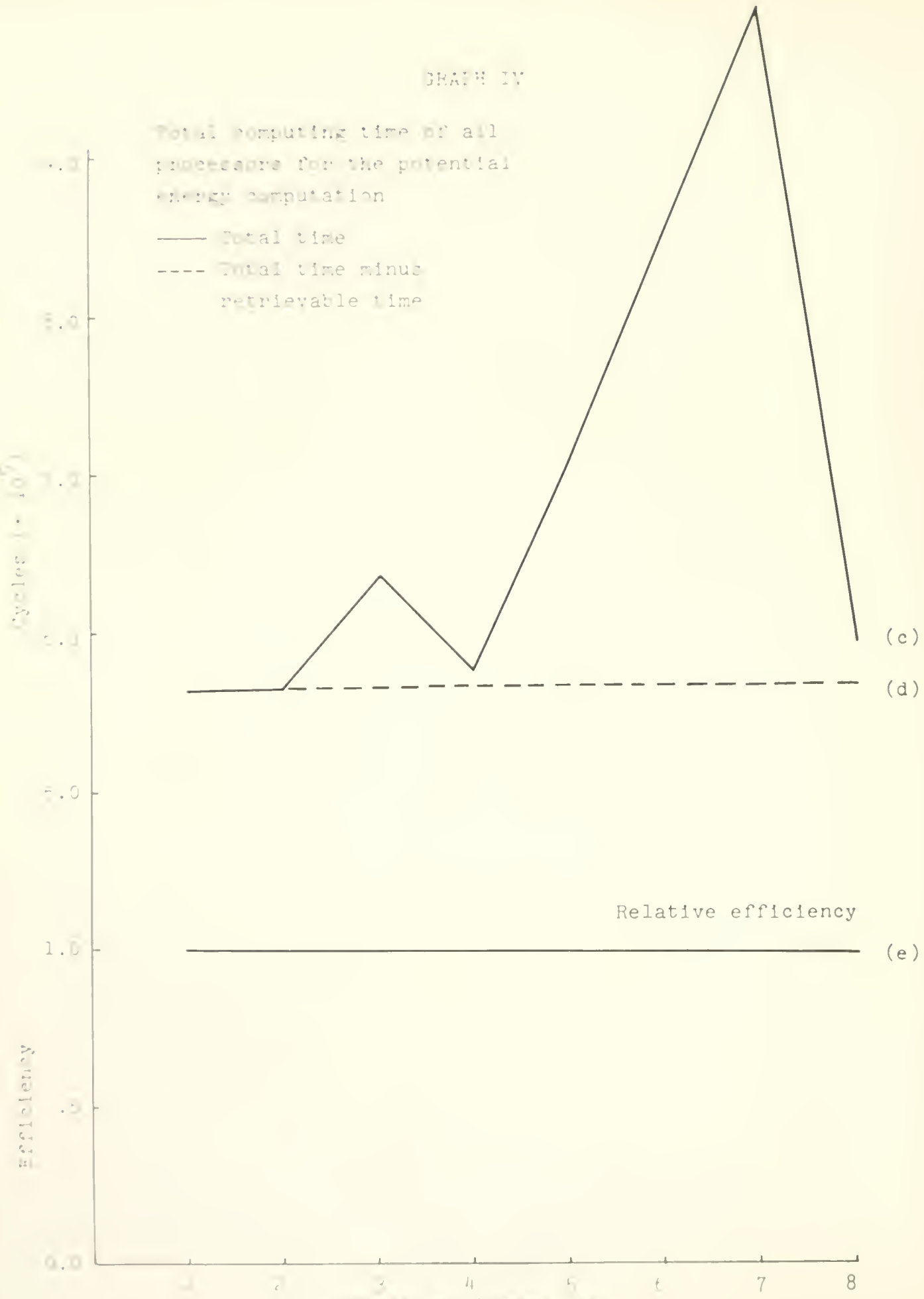
On the next page, curve (c) is total time (column 3), curve (d) is total irretrievable time (column 5), curve (e) is relative efficiency (column 6).

GRAPH III

Completion time for potential energy computation where a fixed number of processors were assigned to the program.



GRAPH IV


Total computing time of all processors for the potential energy computation
Total time
Total time minus retrievable time
(c)
(d)
Relative efficiency
(e)
Cycles (· 10^6)
Efficiency
1.0
0.0
1
2
3
4
5
6
7
8
Number of Processors

N-PARTICLE POTENTIAL ENERGY CALCULATIONS

Runs Under the Simulated Operating System

Entrance Interval	No. of CPU's	Completion Time	Time Ratio
	1	562,963	1.00
80,000	3	293,296	1.92
70,000	4	283,696	1.98
60,000	4	263,091	2.14
50,000	5	243,063	2.32
40,000	5	233,668	2.41
30,000	5	202,836	2.77
20,000	6	193,668	2.90
10,000	7	142,583	3.94
All together	8	73,612	7.66

CHART VII

The following terms are used in this chart:

Entrance interval. The time intervals at which successive new processors were added one at a time to the job.

Time ratio. The ratio of the completion time for one processor to the completion time for a parallel run.

The purpose of this set of runs was to show the effect of late arrival at CPU's on completion time.

N-PARTICLE POTENTIAL ENERGY CALCULATION

Additional Runs under the Simulated Operating System

Eventual No. of CPU's	Completion Time	Total Cycles	No. of Control Points	No. of CPU's in Machine
8	73,713	569,341	4	30
8	74,479	579,878	4	30
8	75,374	580,648	4	30
8	76,454	573,195	4	25
8	81,039	560,805	4	25
8	85,935	554,233	4	30
8	96,018	559,858	4	30
8	98,938	562,807	4	20
8	101,347	565,001	4	25
8	115,418	566,841	4	20
8	121,297	570,718	4	20

CHART VIII

This chart shows the program's performance under the operating system, where the arrival of new CPU's depended on their availability. It can be seen here that the responsiveness of the operating system can mean the difference between a completion time of 74 thousand cycles and a completion time of 121 thousand cycles.

TWO POINT BOUNDARY VALUE COMPUTATION

No. of CPUs	Completion Time	Total Cycles	Eff.	Total Irretrievable Time	Rel. Eff.	No. of Iterations	Q Relative	Q Total	N Average
1	1.462×10^{6}	1.462×10^{6}	1.00	1.462×10^{6}	1.00	7	1.0	1.0	1.0
2	9.488×10^{5}	1.898×10^{6}	.77	1.866×10^{6}	.78	4	1.2	1.2	2.0
3	7.911×10^{5}	2.373×10^{6}	.62	2.261×10^{6}	.65	3	1.2	1.1	2.9
4	6.238×10^{5}	2.495×10^{6}	.58	2.281×10^{6}	.64	2	1.5	1.4	3.7
5	6.748×10^{5}	3.374×10^{6}	.43	2.892×10^{6}	.51	2	1.1	.9	4.3
6	7.434×10^{5}	4.460×10^{6}	.33	3.526×10^{6}	.41	2	.8	.6	4.8
7	5.079×10^{5}	3.555×10^{6}	.41	2.735×10^{6}	.53	1	1.5	1.2	5.4
8	5.966×10^{5}	4.773×10^{6}	.31	3.213×10^{6}	.45	1	1.1	.9	5.4

CHART IX

This chart displays a typical set of runs in which the efficiency did not decrease monotonically with increasing number of CPU's, because another variable (number of iterations to reach the desired precision of the answers) also affected the completion time.

TWO POINT BOUNDARY VALUE COMPUTATION

	No. of CPUs	Completion Time	Total Cycles	Eff.	Total Irretrievable Time	Rel. Eff.	No. of Iterations	Q Relative	Q Total	N Average
Run 1	1	1.645×10^{6}	1.645×10^{6}	1.00	1.645×10^{6}	1.00	8	1.0	1.0	1.0
	2	9.488×10^{5}	1.898×10^{6}	.87	1.865×10^{6}	.88	4	1.5	1.5	2.0
	4	6.238×10^{5}	2.495×10^{6}	.66	2.281×10^{6}	.72	2	1.9	1.7	3.7
	8	5.966×10^{5}	4.773×10^{6}	.35	3.213×10^{6}	.51	1	1.4	1.0	5.4
Run 2	1	3.108×10^{6}	3.108×10^{6}	1.00	3.108×10^{6}	1.00	16	1.0	1.0	1.0
	2	1.715×10^{6}	3.430×10^{6}	.91	3.366×10^{6}	.92	8	1.7	1.5	2.0
	4	1.065×10^{6}	4.260×10^{6}	.73	3.832×10^{6}	.81	4	2.4	2.1	3.6
	8	1.008×10^{6}	8.066×10^{6}	.39	4.945×10^{6}	.63	2	1.9	1.2	4.9

CHART X

This chart shows two independent sets of runs. The total number of iterations (and hence the precision of the answers) for each set of runs was held constant, so that the effect of varying numbers of CPU's on the other variables could be seen.

TWO POINT BOUNDARY VALUE COMPUTATION

Time for One Iteration

No. of CPU's	Single Iteration Time	Total Cycles	Total Cycles Less Retrievable Time
1	3.656×10^5	3.656×10^5	3.656×10^5
2	3.744×10^5	7.488×10^5	7.409×10^5
3	3.855×10^5	1.157×10^6	1.119×10^6
4	4.032×10^5	1.613×10^6	1.506×10^6
5	4.288×10^5	2.144×10^6	1.903×10^6
6	4.631×10^5	2.779×10^6	2.312×10^6
7	5.079×10^5	3.555×10^6	2.735×10^6
8	5.966×10^5	4.773×10^6	3.213×10^6

CHART XI

This table shows the time required for a single iteration through the program. The time <u>increases</u> with an increase in the number of CPU's, but, as is clear from the two preceding charts, the completion time for the whole program still <u>decreases</u>. This is due to the fact that, with more CPU's, <u>fewer</u> iterations are required.

TWO POINT BOUNDARY VALUE COMPUTATION

Runs Under the Simulated Operating System

No. of CPU's	Completion Time	Useful Cycles	No. of Jobs Run Simultaneously	No. of CPU's in Machine
1	1,464,583	1,464,583	4	20
2	775,274	1,531,464	4	20
2	775,576	1,532,058	4	25
2	799,161	1,531,360	4	20
4	449,140	1,548,368	4	20
4	474,347	1,548,230	4	20
8	412,016	1,860,003	4	30
8	413,369	1,870,827	4	25

CHART XII

This chart shows the behavior of the program under the operating system. Late arrival of CPU's was not permitted by the program, but the program did wait a certain amount of time before beginning so that it could accumulate as many CPU's as possible.

MONTE-CARLO COMPUTATION OF POTENTIAL ENERGY

Runs Under the Simulated Operating System

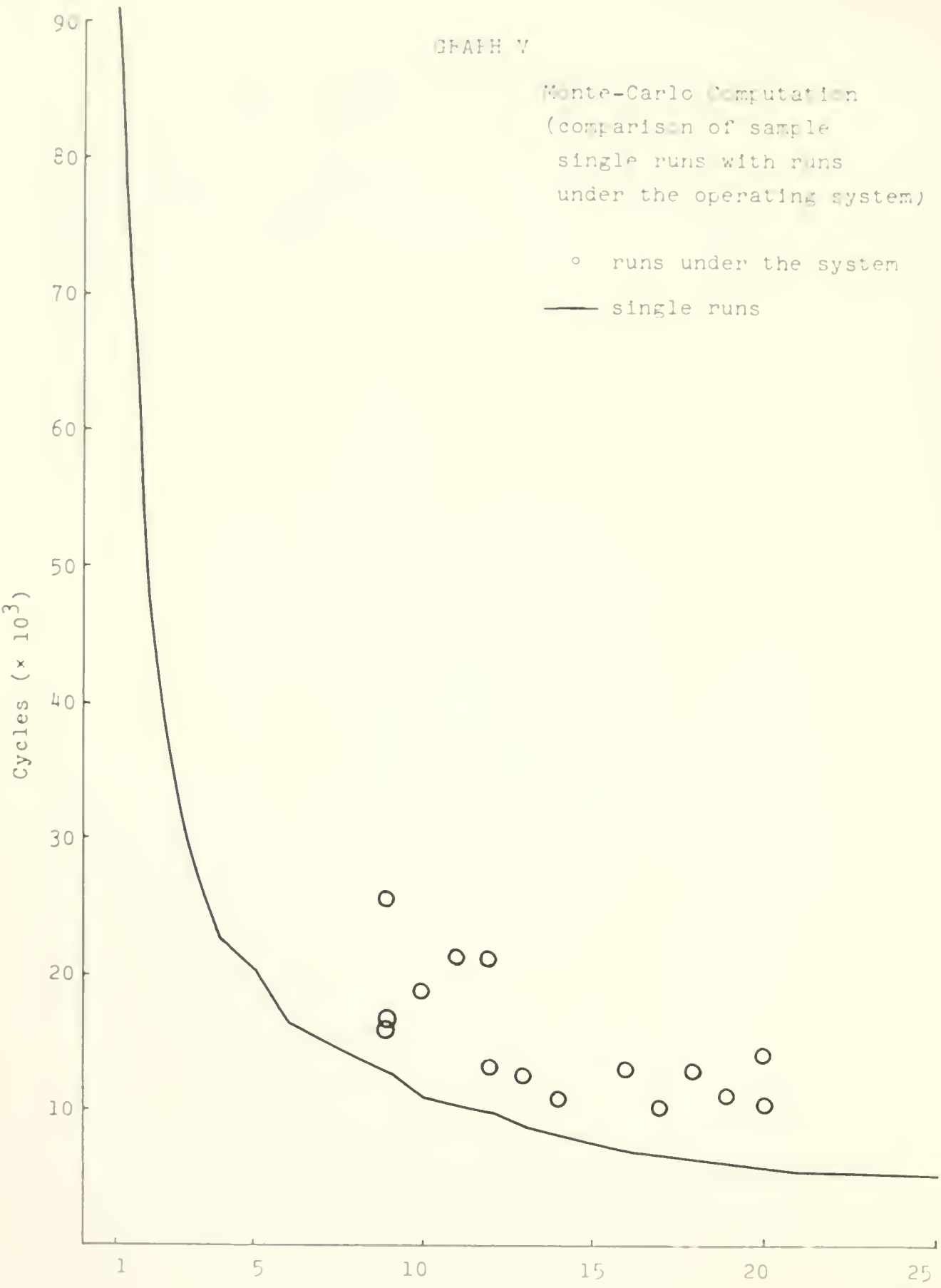
No. of CPU's	Completion Time	Total No. of Cycles	No. of Jobs Run= Simultaneously	No. of CPU's in Machine
8	80,045	171,282	4	20
9	16,344	143,856	4	20
9	25,434	175,289	4	20
9	15,773	142,957	4	25
10	18,872	150,888	4	30
10	56,196	191,336	4	20
11	21,415	167,262	4	25
12	16,279	190,959	4	30
12	16,337	195,840	4	30
12	13,264	159,068	4	25
13	12,706	158,506	4	30
14	10,852	147,378	4	25
16	13,116	151,338	4	20
17	10,145	172,465	4	25
18	12,933	190,743	4	30
19	11,248	146,290	4	25
20	14,359	213,919	4	30
21	21,396	197,465	4	30

CHART XIII

The performance of this program under the operating system shows the effects of late arrival of CPU's and shows that this effect tended to be greater the fewer CPU's there were in the machine.

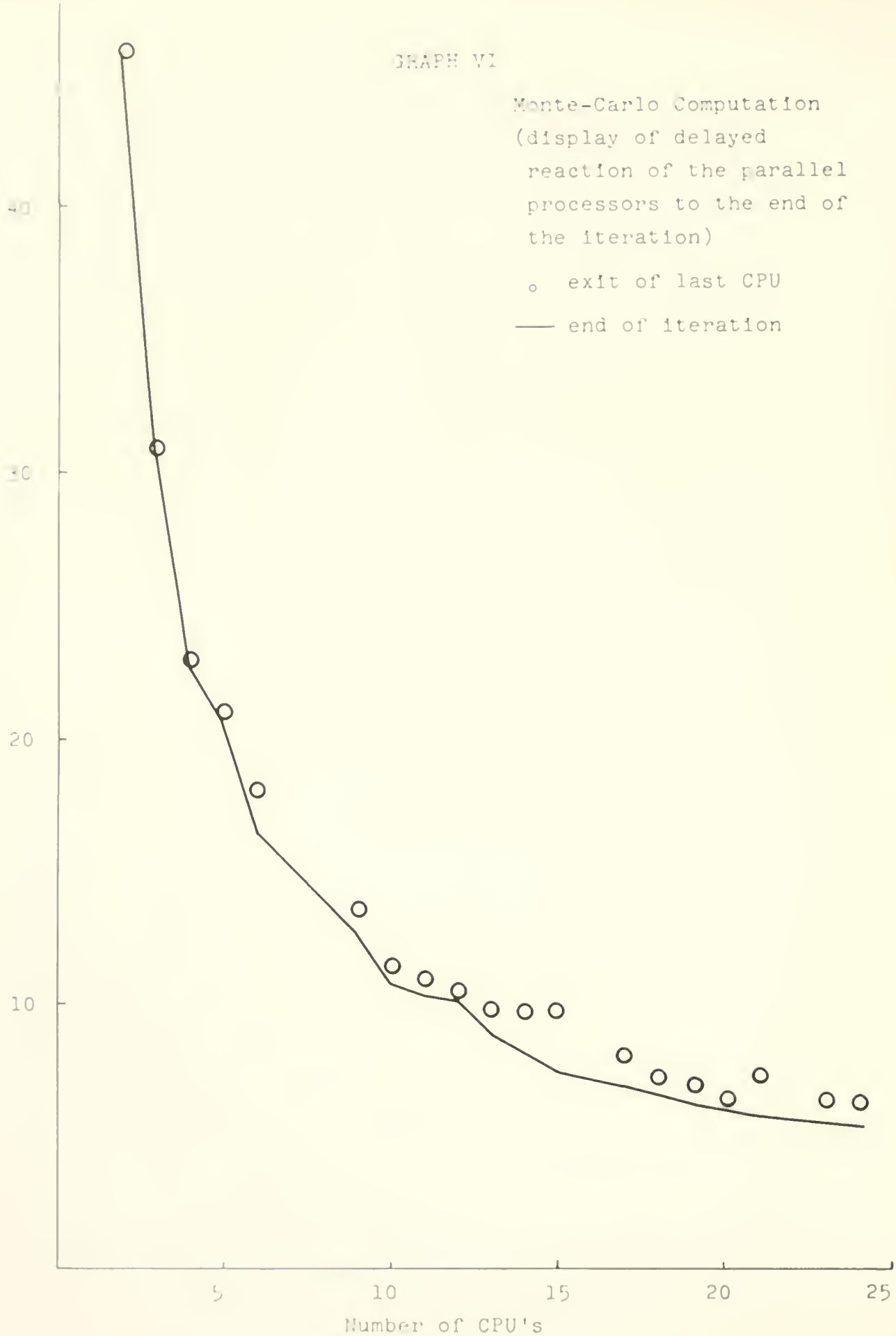
These effects are shown in the graph on the next page which compares the performance of the program when run alone with its performance under the operating system.

GRAPH V


Monte-Carlo Computation
(comparison of sample
single runs with runs
under the operating system)
runs under the system
single runs
Cycles (× 10^3)
90
80
70
60
50
40
30
20
10
1
5
10
15
20
25
Number of CPU's

The following graph shows the time difference between completion of an iteration of the algorithm by the first CPU (which sets a flag to signal the rest of the CPU's to stop) and the time at which the rest of the CPU's acutally stopped.

GRAPH VI


Monte-Carlo Computation
(display of delayed
reaction of the parallel
processors to the end of
the iteration)
o exit of last CPU
—— end of iteration
40
30
20
10
5
10
15
20
25
Number of CPU's

PARALLEL MINIMUM SEARCH

Run Under the Simulated Operating System

No. of CPU's Assigned	Completion Time*	Total Cycles*	Simul. No. of Jobs	No. of CPU's in Machine
2	118	231	4	20
4	99	352	4	20
6	52	297	4	20
10	61	543	5	20
18	42	647	5	25

*In thousands.

CHART XIV

This program was run only under the operating system. It did not permit late arrival of CPU's, and the algorithm was defined only for even numbers of CPU's. As uaual, the completion time decreases with increases in the number of CPU's.

MATRIX MULTIPLY

10 × 10

Run Under the Simulated Operating System

No. of CPU's	Completion Time	Total Cycles	No. of Jobs Run Simultaneously	No. of CPU's in Machine
1	4974	2969	4	20
3	3843	3035	5	20
4	3809	3101	4	20
5	3523	3151	5	25
6	3405	3233	4	20

CHART XV

The small size of the matrix prevented the performance of this program from showing more dramatic gains in speed from increases in the number of CPU's. It does, however illustrate our point that characteristics of the data affect the method of treatment: this size matrix should be handled by only 3 CPU's, since adding more does not result in an appredіable gain in speed.

PARALLEL SORTING ALGORITHM

Run Under the Simulated Operating System

No. of CPU's	Completion Time*	Total Cycles*	No. of Jobs Run Simultaneoulsy	No. of CPUs in Machine
1	136	134	4	05
2	101	199	4	20
3	94	277	4	20
4	94	351	4	20
4	92	358	5	20
5	96	353	4	20
6	89	521	5	25
7	89	608	4	20
8	77	382	4	15
10	87	860	5	25
16	89	1391	5	20
18	113	1526	5	25

*In thousands.

CHART XVI

This program, run under the operating system, shows that (in this case) increases in the number of CPU's eventually results in the completion time's reaching a minimum and thereafter beginning to increase. The program was especially poorly coded.

UNORDERED LIST SEARCH ROUTINE

Run Under the Simulated Operating System

No. of CPU's	Completion Time	Total Cycles	No. of Jobs Run Simultaneously	No. of CPU's in Machine
1	13950	11792	4	15
2	10309	12012	4	15
3				
4	5723	12020	5	25
5	5513	12020	5	20
6	4410	12134	5	25
8	4260	12289	5	20
12	3945	12286	5	25
13	3830	12284	5	25
24	3365	13209	5	25

CHART XVII

This program showed the usual gains in speed with increases in CPU's even when run under the operating system, since there was no restriction on late arrival of CPU's and since neither the algorithm nor the data required any special number of CPU's.

VII. Measurements of the Overall Efficiency of the Simulated Operating System

We now present various figures showing the overall performance of our simulated operating system, as distinct from the performance of particular jobs under the system. Our first chart shows various percentages related to efficiency, as measured after every 100,000 cycles. The trend toward a higher percentage of "useless" time (i.e. time spent in the system or in idle, or in setup) reflects the fact that in this run (which is fairly typical) a job which used a lot of idle and setup time was rolled in. It also is due to the usual presence of programs with "bugs" which sometimes erroneously put all of their CPU's in idle status.

CHART A

Cycles	100000	200000	300000	400000	500000	600000
U/I	.9421	.9705	.9557	.8757	.8106	.7339
U/S	.9271	.9754	.9645	.9619	.9604	.9600
U/U	.8848	.9265	.8971	.8226	.7633	.6939
U	.9372	.9273	.8870	.8094	.7538	.6779
I	.0250	.0263	.0471	.1304	.1873	.2688
S	.0214	.0227	.0363	.0322	.0320	.0283
T	.0163	.0238	.0297	.0281	.0269	.0251

U/I is the percentage of useful cycles to the sum of useful and idle cycles

U/S is the percentage of useful to useful and system

U/U is the percentage of useful to useful and useless

U is the percentage of the total number of executed cycles which are useful

I is the percentage of the total which are idle cycles

S is the percentage of the total which are system cycles

T is the pcertentage of the total which are other cycles usually used for setup time by the jobs

This is a typical run. Note that, due to programmer errors, and the lack of a time limit feature in our simulated system, there is usually a downward trend in the "useful" categories. The increase in idle time (which always occurred) may be due in part to a more significant failing of this type of operating

system. This is one of the reasons for proposing an alternative operating system (see Appendix III).

Our second chart shows the result of using different machine configurations (e.g. 4 and 5 jobs run simultaneously and 5, 10, 15, 20 and 25 CPU's); the typical percentages shown in the chart were obtained.

We make the following comments.

(1) In general, of course, the fewer CPU's in the machine, the busier they are (i.e. the users have fewer to place into idle status). (2) When many undebugged jobs are present, system performance can be very poor (the users of course are responsible for their own mistakes). (3) System time is usually 3 to 5 percent.

Note also that (1) The elapsed time between the request of a task (or virtual CPU) and the entry of the requested CPU into the job was 300-400 cycles when there were CPU's available, and as much as 80,000 cycles at other times.

(2) The number of instructions required to request tasks or virtual CPU's was about 300 cycles. (So, tasks should be longer than 300 cycles.)

CHART B

No. of Jobs Run Simultaneously	5	5	5	4	4	4
No. of CPU's in Machine	25	10	20	15	5	20
U/I	.7920	.9936	.9452	.9700	.8841	.7946
U/S	.8916	.9772	.9162	.9500	.9533	.9188
U/U	.7022	.9450	.8754	.9000	.8233	.7308
U	.7004	.9637	.9262	.9082	.8228	.6907
I	.1874	.9968	.0215	.0272	.1076	.2402
S	.0862	.0227	.0325	.0410	.0413	.0507
T	.0259	.0268	.0199	.0236	.0285	.0184
U/I		.6855	.8968		.9591	.5199
U/S		.9665	.9620		.9478	.9582
U/U		.6506	.8634		.8757	.5014
U		.6502	.8710		.8757	.5007
I		.2985	.0986		.0373	.4636
S		.0229	.0255		.0482	.0219
T		.0284	.0049		.0387	.0138
U/I			.4226			.5199
U/S			.4226			.9582
U/U			.3681			.5014
U			.3671			.5007
I			.5277			.4636
S			.0891			.0219
T			.0161			.0138

The definition of the ratios U/I, U/S, U/U, etc., is the same as in the previous chart, Chart A.

The remaining charts (Charts C and D) show the proportion of system instructions among all instructions executed by all CPU's during runs of various cycle durations. "System idle time" is the total number of instructions executed by processors which were returned to the system by the jobs and could find nothing to do so they idled waiting for a request to come in. This idle time is a measure of the extent to which the system, given the jobs and the memory size it had, could keep itself busy. These figures are merely illustrative; no firm statistical inferences can be drawn from them.

CHART C

SYSTEM AND SYSTEM IDLE TIMES

5 Jobs Run Simultaneously

Duration of the Run in Cycles	No. of CPU's	System Time	System Idle Time
100100	25	3.8%	29.8%
100100	25	5.4%	21.6%
340447	25	2.2%	8.1%
127395	25	7.9%	16.3%
100100	20	8.6%	6.7%
197074	20	3.4%	5.6%
117807	10	2.2%	0.3%
288724	10	2.3%	6.9%

As can be seen, the proportion of system idle time decreases as the number of CPU's in the machine decreases. System time remains rather stable as might be expected. The amount of system idle time is highly variable, depending almost entirely on the configuration of requests and releases. One could try to secure a more stable demand for work by some dynamic rollin/rollout strategy, but to do so is always to gamble that one will get a better configuration next time a request or release is made; our system has few criteria for making rollin/rollout choices.

CHART D

SYSTEM AND SYSTEM IDLE TIMES

4 Jobs Run Simultaneously

Duration of the Run in Cycles	No. of CPU's	System Time	System Idle Time
280000	20	2.1%	1.0%
220000	20	2.7%	2.8%
218106	20	3.6%	7.4%
147100	15	2.0%	0.3%
163710	15	1.6%	0.2%
193467	05	4.0%	0.0%
140405	05	4.6%	0.0%

Note that the system time is about the same with 4 jobs run simultaneously as with 5 (preceding page). But system idle time is much smaller! This is counter-intuitive, since with more jobs run simultaneously the possibility of idle time should decrease. Yet, in every case it is larger with 5 than with 4 jobs run simultaneously.

We see again the tendency for idle time to decrease with decreasing numbers of CPU's. But system time <u>increases</u> slightly as we decrease the number of CPU's.

Some other data concerning system performance are best summarized than presented in detail.

It may be noted that our runs generally simulated execution of between 150 to 200 thousand cycles. At certain moments during this period, varying numbers of CPU's were idle in the system with no requests to answer. Such idle periods could sometimes last as long as 40 to 60 thousand cycles. At other times during the same runs, it could happen that there were substantially many more requests than available CPU's. For runs with 20 to 25 simulated CPU's, twice that number of task-requests might sometimes be posted; for 5 to 15 CPU's, as many as 75 unsatisfied requests might be posted. How long did posted requests have to wait to find a processor? In 20-25 CPU machines, generally no more than 25 thousand cycles; in 5-15 CPU machines, sometimes as long as 60 to 100 thousand cycles. Given the expected run times of our typical programs, this is a considerable wait.

This observed fluctuation in demand for processing leads at times to processors idling (too few requests at that moment) and at other times to the accumulation of numbers of unsatisfied requests. To some extent, of course, this could have been handled by classifying jobs into two categories: those with many requests outstanding at a given moment and those with no requests outstanding. By loading an appropriate mixture of these two kinds of jobs one could hope to keep the system busy. The question of course is by what strategy can such a selection be made?

VIII. Conclusions

The following conclusions, drawn from the data and commentary given above and from our previous report, sum up our experience.

A. It is difficult and unnecessary in parallelizing a program to ferret out every bit of parallelism implicit in it.

B. There do exist large classes of problems which possess naturally parallel structures (perhaps after very slight restructuring). The parallel portions of such problems are, in general, composed of a mixture of the following prime types.

1. Sets of essentially identical operations applied to different subparts of a total data structure as, for example, in the solution of partial differential equations.

2. Similar but not necessarily identical operations performed simultaneously on different subparts of a total data structure.

3. Completely independent operations that are performed at the same time on different data with relatively infrequent communication between the independent program segments.

All of these cases yield with little difficulty to "parallelization" on Athene type computers. This is to say that any program significant portions of which have one of the structures described above can readily be converted to a reasonable parallel version. Note that only the parallelism of the first above type is useful for Illiac type computers. In this connection we may cite the following statement:

"It is not too difficult to see why Illiac IV performs so poorly (i.e. in comparison with its performance on PDE problems) on the table look-up problem. The 256 separate memories for the 256 Program Elements are a distinct hardship. Better than 80% of the time required to accomplish the problem is spent in shuffling the table entries..."[18]

These considerations make it clear that computers of the Illiac IV type are basically special purpose machines for which the complete power of the parallel machine structure is usable only for certain particular problems.

C. Although it is not absolutely necessary to have an elaborate operating system for reasonably efficient parallel multi-processing, it is very important to have some sort of assignment algorithm that produces a reasonable distribution of processors among the various jobs taking due cognizance of the ability of the individual jobs to utilize assigned CPU's. (See, for example, the data in Charts I and IV.)

D. Though not radically different from ordinary serial programming languages, the parallel languages that have been devised appear to be adequate for efficient parallel programming. Moreover, it is probably feasible now to write compilers which can detect certain types of parallelism automatically.

E. Parallel computers of the Athene type proposed can be efficiently utilized and should prove successful when constructed.

---

[18] McIntyre, D., "The Table Lookup Problem Revisited," 4/26/68 Illiac IV Doc. #183.

Bibliography

1. Schwartz, J. T., "Large Parallel Computers," Journal of the ACM, January, 1966.
2. Draughon, E., Grishman, R., Schwartz, J., and Stein, A., Programming Considerations for Parallel Computers, Courant Institute Report IMM 362, November, 1967.
3. Frances, J. G., "The QR Transformation," Computer Journal 4, Part I, 1961, p. 265-271, Part II, 1962, p. 332-345.
4. Murray, Richard, "Parallel QREIGEN Subroutine," NYU Master's Thesis, June, 1967.
5. Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors," Proc. IEEE, Vol. 54, p. 1889-1901, Dec. 1966.
6. McIntyre, D., "The Table Lookup Method Revisited," Illiac IV Doc. # 183, April 26, 1968.
7. Cocke, J. and Schwartz, J. T., Programming Languages and Their Compilers, Courant Institute Preliminary Notes, 1970.

Appendix I. Automatic Detection of Parallelism.

As was indicated in our earlier paper[19] a true parallel compiler, i.e., one able to detect and exploit the parallelism in serially written programs, is desirable if not indispensable.

At the time of writing of [1] (1965) it was not clear how to go about this task. Recently, however, work done on optimization of compiled code (Cf. particularly the work of Dr. J. Cocke and his group)[20] by methods which investigate the topology of compiler source programs has clarified the problems of detecting parallelism. Study of these methods shows that similar devices allow the independence of certain operations to be ascertained, thus allowing execution in parallel. The detailed working-out of the programme is a desirable path for future work.

---

[19] [1], p. 29.

[20] See Cocke, John, and Schwartz, J. T., Programming Languages and Their Compilers, Preliminary Notes, New York University, 1970.

Appendix II

Features of the Illiac IV Parallel Language, TRANQUIL

I. The main features of interest in the TRANQUIL language are its method of handling simultaneous control; its handling of private storage; its elegant way of generating variable-length lists; and some comparatively minor conveniences. The language introduces only one explicit parallel instruction, SIM, which has the effect of executing the statements within its range (either a single statement or a BEGIN-END block) in parallel. SIM is thus analogous to our REQUEST command, but makes it unnecessary to issue a RELEASE explicitly at the end of each parallel segment. SIM also causes the automatic generation of code to check that all the parallel tasks are completed before the execution of the program following the SIM range begins. While SIM is somewhat easier to use than our series of parallel commands, it lacks some of their flexibility (e.g., it does not provide the ability to specify a varying number of processors for the execution of parallel segments). The SIM command allows all variables to be treated within its range as private, the variables resuming their normal public character outside of the SIM range. While this powerful feature (semi-private storage) can be quite convenient in the handling of variables in parallel, it does have the disadvantage of not allowing processor inter-communication during parallel execution. Such inter-communication is easily specified in PFORTRAN. However, the convenience of the SIM concept argues strongly for its inclusion in parallel languages.

TRANQUIL allows the simultaneous changing of variables during parallel loop trasersal, allows the specification of loop variable values in list form rather than in the restricted DO form of FORTRAN; and makes it possible to define these lists as sets (in the mathematical sense) of varying length and with various structures (e.g., the Cartesian product set of two sets can be generated in an elegant way). All of these features are quite useful, especially for the solution of complex matrix problems; however, the features they provide can be omitted without causing too much difficulty.

In TRANQUIL, sets of n-tuples can be defined in any or all of the following ways.

1. Explicitly by listing their members
2. By a DO-loop-like specification
3. By a DATA-statement-like specification
4. By concatenation
5. As the reversal of another set
6. By conditional operations on other sets
7. By pairing (i.e., a set can be formed by taking pairs of two other sets)
8. By the Cartesian product of two sets
9. By combinations of all of the above.

Sets defined in this manner can then be used to control iterative loops as indicated above. Excepting the Cartesian product construction, however, it is not clear how useful all these constructions really are for parallel programming.

Comparing TRANQUIL with the PFORTRAN language of the present study, and excepting the semi-private storage concept and the index list construction, provided by TRANQUIL, it may however be doubted that any of these features will have significant relative advantage in the conversion of uniprocessor programs to parallel ones, or, for that matter, the initial programming of parallel programs. Nevertheless, the idea of semi-private storage is a significant advance, especially as its invocation has been made implicit in TRANQUIL. However, it may not be easy to make the frequent use of semi-private storage very efficient (though perhaps an optimizing scan of the range of the SIM statements could be used to minimize the extent to which private storage had to be allocated).

Appendix III.

An Alternative Control Algorithm

Our present operating system algorithm represents a parallel-processor analog of the ordinary run-to-completion batch system. This control program shows a number of deficiencies. Jobs may not get the precise number of CPU's that they request when they request them. One job can monopolize most of the CPU's, delaying the completion of other jobs. And conditions of too few task requests (idle CPU's) or too many task requests have to be combatted with frequent roll-in/roll-out procedures.

To cure these deficiencies, an operating system incorporating ideas taken from single-processor multiprogramming systems might be desirable. In such a system, each task on each task list would be treated by the system as a "pseudo control point". The real CPU's would be interrupted n milliseconds of run on a given problem and transferred to the next set of pseudo control points until all loaded jobs have been run for n milliseconds. As new tasks were added, they would become "pseudo-control points and would begin execution in a short time.

Thus, for example, if there were a total of X tasks on all task lists and there are Y real CPU's, the first Y tasks would be executed for n milliseconds, then the next Y tasks, and so on until all X have been executed, then the process would repeat. The number X of tasks pending may change at any time without interrupting this process. Priorities can be implemented by

allowing tasks stemming from the highest priority job to run for a slightly longer period. This procedure allows all jobs to move along smoothly to completion, always getting the number of CPU's they request. Moreover, the large percentages of "system idle time" shown in Chart C would be zero.

If an operating system of this description were to be implemented, dual sets of registers might be provided for each CPU, so that one set could operate while the other is being saved in main memory and refilled with the next set of registers needed. Note also that if the hardware were such that only a limited number of memory locations to which the fundamental RAD instruction may be addressed are available (and our experience has shown that only a limited number, perhaps 1000, are needed), then it would be convenient to have an instruction which could interchange the contents of any one of these locations with another memory location (to insure that a sufficient number of specialized locations is always available).